中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2020K8009926006 姓名： 游昆霖 专业： 计算机科学与技术

实验序号： 2 实验名称： 简单功能型处理器设计

注1：撰写此Word格式实验报告后以PDF格式保存在~/COD-Lab/reports目录下。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为1至4的阿拉伯数字。例如：prj1.pdf。PDF文件大小应控制在5MB以内。此外，实验项目5包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目5讲义。

注2：使用git add及git commit命令将实验报告PDF文件添加到本地仓库master分支，并通过git push推送到GitLab远程仓库master分支（具体命令详见实验报告）。

注3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

1. 逻辑电路结构与仿真波形的截图及说明（比如关键RTL代码段{包含注释}及其对应的逻辑电路结构图、相应信号的仿真波形和信号变化的说明等）

说明：单周期处理器部分设计共包含四个模块文件，其中reg\_file直接复用prj1所完成即可，alu需新增3个操作，同时新加入shifter和simple\_cpu模块。多周期处理器在单周期的基础上新增了状态机和部分寄存器。以下只对处理器给出逻辑电路结构图和仿真波形说明。

1. alu.v 代码新增操作说明

|  |
| --- |
| //new ALUop  `define XOR 3'b100  `define NOR 3'b101  `define SLTU 3'b011  ...  //new wire virables for direct results  **wire** **[**`DATA\_WIDTH **-**1 **:**0**]** R\_XOR**;**  **wire** **[**`DATA\_WIDTH **-**1 **:**0**]** R\_NOR**;**  **wire** **[**`DATA\_WIDTH **-**1 **:**0**]** R\_SLTU**;**  ...  **assign** R\_XOR **=** A **^** B**;**  **assign** R\_NOR **=** **~**R\_OR**;**  ...  **assign** Inverse **=** ALUop**[**1**]** **&** **(**ALUop**[**2**]** **|** ALUop**[**0**]);** //sub or slt or sltu  **assign** A\_tmp **=** **{**1'b0**,**A**};**  **assign** B\_tmp **=** **{**1'b0**,** Inverse **?** **~**B **:** B **}** **+** Inverse **;** //note that the inverse result exclude the highest bit    //result of OP ADD / SUB / SLT  **assign** **{**flag **,** R\_ADD**}** **=** A\_tmp **+** B\_tmp**;**  **assign** R\_SLTU**[**`DATA\_WIDTH**-**1 **:** 0**]** **=** **{**31'b0**,** **~**flag**};** |

新增的三个操作中XOR异或和NOR非或操作基于其语义进行操作即可。SLTU无符号数比较操作只需将A、B增加1位表示为补码，进行减法操作，并根据结果符号位判断即可。

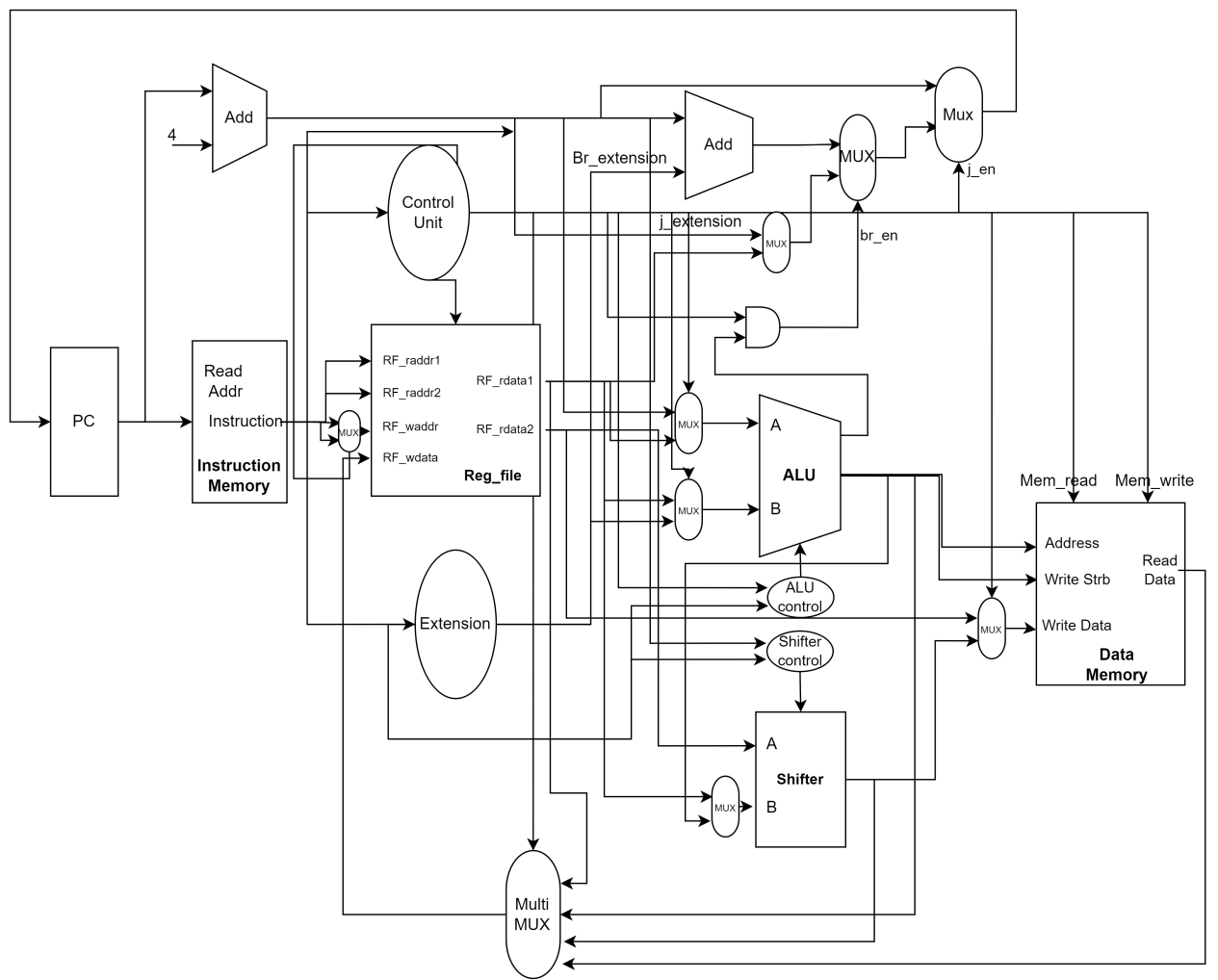
1. shifter.v 代码说明

|  |
| --- |
| //Shiftop  `define SLL 2'b00  `define SRL 2'b10 //set the word right logical  `define SRA 2'b11 //set the word right arithmetic  //wire variable for direct results  //SRA: care space left by SRA,so do not use simple shift  **wire** **[**`DATA\_WIDTH **-** 1**:**0**]** R\_SLL**;**  **wire** **[**`DATA\_WIDTH **-** 1**:**0**]** R\_SRL**;**  **wire** **[**`DATA\_WIDTH **-** 1**:**0**]** R\_SRA**;**  **wire** **[**63**:**0**]** SRA\_tmp**;**  **assign** R\_SLL **=** A **<<** B**[**4**:**0**];**  **assign** R\_SRL **=** A **>>** B**[**4**:**0**];**  **assign** SRA\_tmp **=** **{{**32**{**A**[**31**]}},**A**}** **>>** B**[**4**:**0**];**  **assign** R\_SRA **=** SRA\_tmp**[**31**:**0**];**  **assign** Result **=** **({**32**{**Shiftop **==** `SLL**}}** **&** R\_SLL**)** **|**  **({**32**{**Shiftop **==** `SRL**}}** **&** R\_SRL**)** **|**  **({**32**{**Shiftop **==** `SRA**}}** **&** R\_SRA**)** **;** |

逻辑移位按照语义进行操作即可，对于算数右移，由于高位补充的结果依赖于其符号位，因此先将A符号拓展为64位，再截取移位后的低32位作为算数右移结果即可。

1. 单周期simple\_cpu.v 代码说明

逻辑结构图：其中只含有PC+4 和PC+br\_extension 两个加法器和ALU，只含有Shifter一个移位器。



省略变量端口声明部分，其余根据逻辑相关程度，可分为ID译码部分，ALU部分，Shifter部分，Reg\_file部分，PC部分和MEM访存部分6个部分，以下将分部分给出代码逻辑说明，并就关键部分给出仿真波形说明。

1. ID译码部分

|  |
| --- |
| //Analyse Instruction code  **assign** opcode **=** Instruction**[**31**:**26**];**  **assign** rs **=** Instruction**[**25**:**21**];**  **assign** rt **=** Instruction**[**20**:**16**];**  **assign** rd **=** Instruction**[**15**:**11**];**  **assign** sa **=** Instruction**[**10**:**6**];**  **assign** func **=** Instruction**[**5**:**0**];**  **assign** imm **=** Instruction**[**15**:**0**];**  **assign** index **=** Instruction**[**25**:**0**];**    //extension prepared for operation  **assign** zero\_extension **=** **{** 16'b0 **,** imm **};**  **assign** sign\_extension **=** **{** **{**16**{**imm**[**15**]}}** **,**imm **};**  **assign** br\_extension **=** **{** **{**14**{**imm**[**15**]}}** **,**imm **,** 2'b0 **};** //use in Branch  **assign** j\_extension **=** **{** PC4**[**31**:**28**]** **,** index **,** 2'b0 **};** //use in Jump |

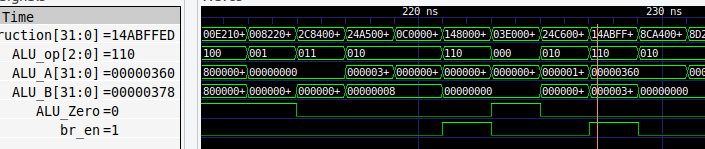
通过将指令集进行归纳，可得到Instruction的译码各信号对应位置，进行连接即可。同时根据不同指令的操作需求，可事先准备好4类拓展，即imm的零拓展和符号拓展，以及关于分支和跳转指令的拓展。

1. ALU部分

|  |
| --- |
| //signals connected to ALU  //All type below can differ from others  //Operation related to ALU list as : Type (op\_A op\_B) num : feature  //R calc (rs rt) 8 : opcode[5:0]=000000 & func[5]=1  //I calc (rs imm) 6 : opcode[5:3]=001 & (~&opcode[2:0]) diff from lui  //REGIMM (rs 0) 2 : opcode[5:0]=000001 BGEZ BLTZ GPR[rs]?0 using SLT(sign)  //Branch (rs rt) 2 : opcode[5:1]=00010  //(rs 0) 2 : opcode[5:1]=00011 using SUB and test ZERO or signflag  //load and store (rs imm) 12 : opcode[5]=1 using ADD  //base(rs)+offset(imm)  //JAL (PC\_reg 8) 1: opcode[5:0]=000011  //JALR (PC\_reg 8) 1: opcode[5:0]=000000 & func[5:1]=001001  //Operation not related to ALU list:  //shifter 6 : opcode[5:0]=000000 & func[5:3]=000  //jump 2 : J opcode[5:0]=000010  //JR opcode[5:0]=000000 & func[5:0]=001000  //lui 1 : opcode[5:0]=001111  //move (rt 0) 2 : opcode[5:0]=000000 & func[5:1]=00101  //move rs to rd,  //we can use ALU(ADD/SUB) or no, and we should care the condition about Reg\_wen  **assign** ALU\_op **=** **(~|**opcode**[**5**:**0**])** **&** func**[**5**]==**1'b1 **?** **(**func**[**3**:**2**]==**2'b00 **?** **{**func**[**1**],**2'b10**}** **:**  func**[**3**:**2**]==**2'b01 **?** **{**func**[**1**],**1'b0**,**func**[**0**]}** **:**  func**[**3**:**2**]==**2'b10 **?** **{~**func**[**0**],**2'b11**}** **:**  0 **)** **:**  opcode**[**5**:**3**]==**3'b001 **&** **(~&**opcode**[**2**:**0**])** **?** **(**opcode**[**2**:**1**]==**2'b00 **?** **{**opcode**[**1**],**2'b10**}** **:**  opcode**[**2**]==**1'b1 **?** **{**opcode**[**1**],**1'b0**,**opcode**[**0**]}** **:**  opcode**[**2**:**1**]==**2'b01 **?** **{~**opcode**[**0**],**2'b11**}** **:**  0 **)** **:**  opcode**[**5**:**0**]==**6'b000001 **?** 3'b111 **:**  opcode**[**5**:**2**]==**4'b0001 **?** 3'b110 **:**  opcode**[**5**]** **==**1'b1 **?** 3'b010 **:**  opcode**[**5**:**0**]==**6'b000011 **|** **((~|**opcode**[**5**:**0**])** **&** func**[**5**:**0**]==** 6'b001001**)** **?** 3'b010 **:**  0**;** //default  **assign** ALU\_A **=** opcode**[**5**:**0**]==**6'b000011 **|** **((~|**opcode**[**5**:**0**])** **&** func**[**5**:**0**]==**6'b001001**)** **?** PC **:**  RF\_rdata1**;**  //ALU\_B: 4 / 8/ rt / 0 / sign\_extend(imm) /zero\_extend(imm)  //imm\_extension: I\_calc 6 + load and store 12  //zero\_extension : ANDI ORI XORI opcode[2]=1  //sign\_extension : ADDIU SLTI SLTIU opcode[2]=0 load and store: opcode[5]==1  **assign** ALU\_B **=** opcode**[**5**:**3**]==**3'b001 **&** **(~&**opcode**[**2**:**0**])** **?** **(** opcode**[**2**]** **?** zero\_extension **:** sign\_extension **):**  opcode**[**5**]** **?** sign\_extension **:**  opcode**[**5**:**0**]==**6'b000001 **?** 32'b0 **:**  opcode**[**5**:**0**]==**6'b000011 **|** **((~|**opcode**[**5**:**0**])** **&** func**[**5**:**0**]==** 6'b001001**) ?** 32'd8 **:**  RF\_rdata2**;**    alu alu\_inst**(**  **.**A**(**ALU\_A**),**  **.**B**(**ALU\_B**),**  **.**ALUop**(**ALU\_op**),**  **.**Overflow**(**ALU\_Overflow**),**  **.**CarryOut**(**ALU\_CarryOut**),**  **.**Zero**(**ALU\_Zero**),**  **.**Result**(**ALU\_Result**)**  **);** |

ALU部分需要根据指令译码得到的结果确定操作类型以及操作数。不同类型指令的条件特征、对应操作数如代码注释所示，且不同类别的特征互斥，并可唯一确定该类指令。每一类中再根据funct和opcode对应字段得到具体操作类型（优化见第三部分）。特别的，REGIMM使用SLT操作，而其余branch类指令使用SUB操作。

相应波形说明：



光标所示位置为BNE指令，ALU使用SUB操作，所得结果ALU\_Zero不为0，表示两寄存器数不等，因此进行跳转。

1. Shifter部分

|  |
| --- |
| //signals connected to Shifter  //Operation related to Shifter list: 6 + 2(swl swr)  //B width is 5  //R shift: opcode[5:0]=000000 & funct[5:3]=000  //xxx (rt sa) 3: fun[2]=0  //xxxv (rt rs) 3: fun[2]=1  //Swl/swr: opcode[5:0]=101x10  //swl/swr (rt swl\_shifter)  **assign** Shifter\_op **=** **(~|**opcode**[**5**:**0**])&(~|**func**[**5**:**3**])** **?** func**[**1**:**0**]** **:**  **{**opcode**[**5**:**3**],**opcode**[**1**:**0**]}** **==** 5'b10110 **?** **{~**opcode**[**2**],**1'b0**}** **:**  0**;**  **assign** Shifter\_A **=** RF\_rdata2**;**  **assign** Shifter\_B **=** **(~|**opcode**[**5**:**0**])&(~|**func**[**5**:**3**])** **?** **(**func**[**2**]** **?** RF\_rdata1**[**4**:**0**]** **:** sa**):**  **{**opcode**[**5**:**3**],**opcode**[**1**:**0**]}** **==** 5'b10110 **?** **(**opcode**[**2**]** **?** swr\_shifter **:** swl\_shifter**)** **:**  0**;**    shifter shifter\_inst**(**  **.**A**(**Shifter\_A**),**  **.**B**(**Shifter\_B**),**  **.**Shiftop**(**Shifter\_op**),**  **.**Result**(**Shifter\_Result**)**  **);** |

Shifter部分主要支持两种类型，分别为I-type中的Shifter类型指令和Store类型中的swl和swr，相应特征和操作数如代码注释所示。其中swl和swx的移动位数根据ALU\_Result确定。

1. Reg\_file部分

|  |
| --- |
| //signals connected to Reg\_file  **assign** RF\_raddr1 **=** rs**;**  **assign** RF\_raddr2 **=** rt**;**  //Operations related to RF\_wen 32  //Type Num addr data feature  //R calc 8 rd alu\_re opcode[5:0]=000000 & func[5]=1  //I calc 6 rt alu\_re opcode[5:3]=001 & (~&opcode[2:0])  //load 7 rt loaddata opcode[5:3]=100  //shift 6 rd shift\_re opcode[5:0]=000000 & func[5:3]=000  //JAL 1 31 PC8(alu\_re) opcode[5:0]=000011  //JALR 1 rd PC8(alu\_re) opcode[5:0]=000000 & func[5:0]=001001  //LUI 1 rt imm||0^16 opcode[5:0]=001111  //MOV 2 rd rdata1 opcode[5:0]=000000 & func[5:1]=00101  //mov is only use condition to refresh GPR, br or jump refresh PC, so we should consider whether is 0  //Operation not related to RF\_wen 13  //Type Num feature  //REGIMM 2 opcode[5:0]=000001  //Branch 4 opcode[5:2]=0001  //store 5 opcode[5:3]=101  //J 1 opcode[5:0]=000010  //JR 1 opcode[5:0]=000000 & func[5:0]=001000  **assign** RF\_wen **=** **(**opcode**[**5**:**0**]==**6'b000000 **&** func**[**5**]==**1'b1**)** **|**  **(**opcode**[**5**:**3**]==**3'b001 **&** **(~&**opcode**[**2**:**0**]))** **|**  **(**opcode**[**5**:**3**]==**3'b100**)** **|**  **(**opcode**[**5**:**0**]==**6'b000000 **&** func**[**5**:**3**]==**3'b000**)** **|**  **(**opcode**[**5**:**0**]==**6'b000011**)** **|**  **(**opcode**[**5**:**0**]==**6'b000000 **&** func**[**5**:**0**]==**6'b001001**)** **|**  **(**opcode**[**5**:**0**]==**6'b001111**)** **|**  **((**opcode**[**5**:**0**]==**6'b000000 **&** func**[**5**:**1**]==**5'b00101**)** **&** **((**func**[**0**]** **&** **(|**RF\_rdata2**))|(~**func**[**0**]** **&** **(~|**RF\_rdata2**))))** **;**  **assign** RF\_waddr **=** **(**opcode**[**5**:**3**]==**3'b100 **|** opcode**[**5**:**3**]==**3'b001 **)** **?** rt **:**  **(**opcode**[**5**:**0**]==**6'b000011**)** **?** 5'd31 **:**  rd **;**  **assign** RF\_wdata **=** **((**opcode**[**5**:**0**]==**6'b000000 **&** func**[**5**:**1**]==**5'b00101**)** **&** **((**func**[**0**]** **&** **(|**RF\_rdata2**))|(~**func**[**0**]** **&** **(~|**RF\_rdata2**))))** **?** RF\_rdata1 **:**  **(**opcode**[**5**:**3**]==**3'b100**)** **?** load\_data **:** //define below  **(**opcode**[**5**:**0**]==**6'b000000 **&** func**[**5**:**3**]==**3'b000**)** **?** Shifter\_Result **:**  **(**opcode**[**5**:**0**]==**6'b001111**) ?** **{**imm**,**16'b0**}** **:**  ALU\_Result **;**    reg\_file reg\_file\_inst**(**  **.**clk**(**clk**),**  **.**raddr1**(**RF\_raddr1**),**  **.**raddr2**(**RF\_raddr2**),**  **.**rdata1**(**RF\_rdata1**),**  **.**rdata2**(**RF\_rdata2**),**  **.**waddr**(**RF\_waddr**),**  **.**wdata**(**RF\_wdata**),**  **.**wen**(**RF\_wen**)**  **);** |

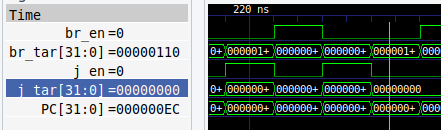
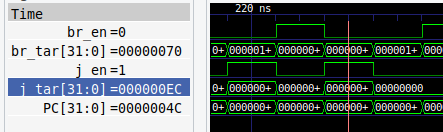
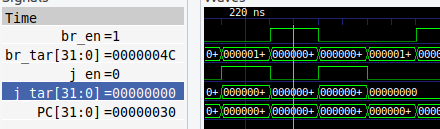
Reg\_file部分除了完成内存读之外，内存写部分主要包含对运算calc、内存读load、寄存器数据移动MOV和跳转Link相关指令。指令类别对应特征及写数据如注释所示。

1. PC部分

|  |
| --- |
| //signals connected to PC  //branch condition test zero signflag and overflow:  //branch target is same  //BEQ,BNE,BLEZ,BGTZ (using SUB): opcode[5:2]=0001 opcode[1:0]=00,01,10,11  //BLTZ,BGEZ (using SLT): opcode[5:0]=000001 rt[0]=0 1  //jump has no condition  //jump target differ:  //J JAL : j\_extension opcode[5:1] =00001  //JR JALR : rs opcode[5:0] = 000000 & func[5:1]=00100    **assign** br\_en **=** **(** **(**opcode**[**5**:**2**]==**4'b0001**)** **&** **(** **(** **(**opcode**[**1**:**0**]==**2'b00**)** **&** ALU\_Zero**)**  **|**  **(** **(**opcode**[**1**:**0**]==**2'b01**)** **&** **~**ALU\_Zero**)** **|**  **(** **(**opcode**[**1**:**0**]==**2'b10**)** **&** **(**ALU\_Zero **|** **(**ALU\_Overflow **^** ALU\_Result**[**31**]** **)))** **|**  **(** **(**opcode**[**1**:**0**]==**2'b11**)** **&** **(~**ALU\_Zero **&** **(**ALU\_Overflow **~^** ALU\_Result**[**31**])))** **)**  **)** **|**  **(** **(**opcode**[**5**:**0**]==**6'b000001**)&** **((~**rt**[**0**]** **&** ALU\_Result**[**0**])** **|**  **(**rt**[**0**]** **&** **~**ALU\_Result**[**0**])** **)**  **)** **;**  **assign** br\_tar **=** PC4 **+** br\_extension **;**  **assign** j\_en **=** **(**opcode**[**5**:**1**]==**5'b00001**)** **|** **((~|**opcode**[**5**:**0**])&(**func**[**5**:**1**]==**5'b00100**))** **;**  **assign** j\_tar **=** **({**32**{(**opcode**[**5**:**1**]==**5'b00001**)}}** **&** j\_extension **)** **|**  **({**32**{((~|**opcode**[**5**:**0**])&(**func**[**5**:**1**]==**5'b00100**))}}** **&** RF\_rdata1**);**    **always** **@** **(posedge** clk**)** **begin**  **if** **(**rst**)** **begin**  PC **<=** 32'b0**;**  **end**  **else** **begin**  PC **<=** br\_en **?** br\_tar **:** j\_en **?** j\_tar **:** PC4 **;**  **end**  **end** |

PC更新主要可分为三类：顺序更新、条件跳转和无条件跳转。其中条件跳转需要结合ALU运算结果，同时因为mips指令集分支延迟槽的存在，br\_tar其中一个加数为PC+4。

相应波形说明：



以上分别为发生条件跳转和无条件跳转时PC的更新情况，当对应使能信号拉高时，PC更新为对应目标。

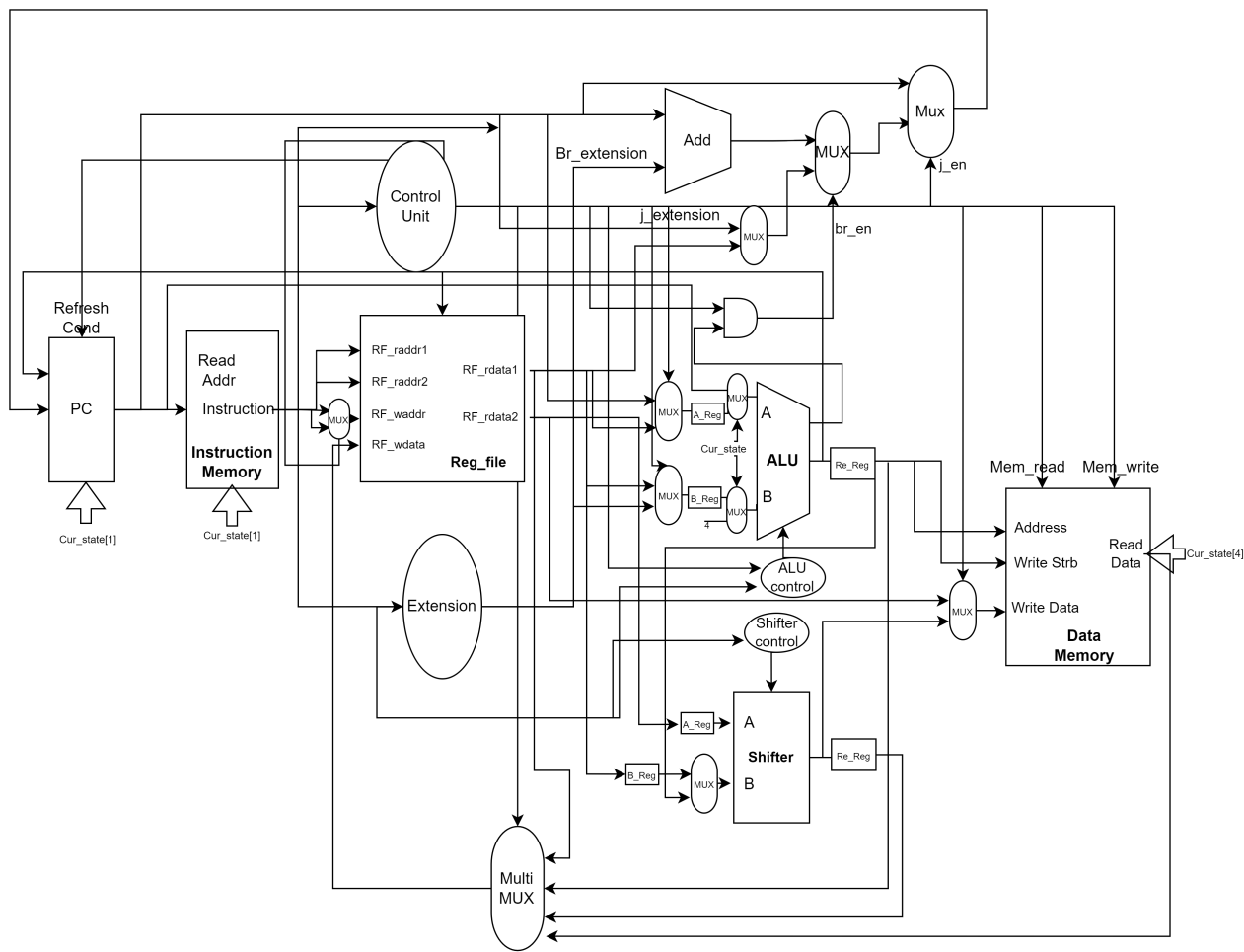
1. MEM访存部分

|  |
| --- |
| //signals connected to memory (load and store)  //load opcode[5:3] = 3'b100 store opcode[5:3]=101 using Little\_endian  //mem control  **assign** MemRead **=** opcode**[**5**]** **&** **~**opcode**[**3**];** //100  **assign** MemWrite **=** opcode**[**5**]** **&** opcode**[**3**];** //101  **assign** Address **=** **{**ALU\_Result**[**31**:**2**]** **,** 2'b0**};**    //load data mem -> rt  //byte means vaddr/ALU\_result[2:0] memword means Read\_data  **assign** lb\_data **=**  **({** 32**{~**ALU\_Result**[**1**]** **&** **~**ALU\_Result**[**0**]}** **}** **&** **{** **{**24**{**Read\_data**[** 7**]}}** **,** Read\_data**[** 7**:** 0**]** **}** **)|**  **({** 32**{~**ALU\_Result**[**1**]** **&** ALU\_Result**[**0**]}** **}** **&** **{** **{**24**{**Read\_data**[**15**]}}** **,** Read\_data**[**15**:** 8**]** **}** **)|**  **({** 32**{** ALU\_Result**[**1**]** **&** **~**ALU\_Result**[**0**]}** **}** **&** **{** **{**24**{**Read\_data**[**23**]}}** **,** Read\_data**[**23**:**16**]** **}** **)|**  **({** 32**{** ALU\_Result**[**1**]** **&** ALU\_Result**[**0**]}** **}** **&** **{** **{**24**{**Read\_data**[**31**]}}** **,** Read\_data**[**31**:**24**]** **}** **);**  **assign** lh\_data **=** **({** 32**{~**ALU\_Result**[**1**]}** **}** **&** **{** **{**16**{**Read\_data**[**15**]}}** **,** Read\_data**[**15**:** 0**]** **})|**  **({** 32**{** ALU\_Result**[**1**]}** **}** **&** **{** **{**16**{**Read\_data**[**31**]}}** **,** Read\_data**[**31**:**16**]** **}** **);**  **assign** lw\_data **=** Read\_data **;**  **assign** lbu\_data **=** **{**24'b0 **,** lb\_data**[** 7**:**0**]}** **;**  **assign** lhu\_data **=** **{**16'b0 **,** lh\_data**[**15**:**0**]}** **;**  **assign** lwl\_data **=**  **({** 32**{~**ALU\_Result**[**1**]** **&** **~**ALU\_Result**[**0**]}** **}** **&** **{** Read\_data**[** 7**:** 0**]** **,** RF\_rdata2**[**23**:** 0**]** **}** **)|**  **({** 32**{~**ALU\_Result**[**1**]** **&** ALU\_Result**[**0**]}** **}** **&** **{** Read\_data**[**15**:** 0**]** **,** RF\_rdata2**[**15**:** 0**]** **}** **)|**  **({** 32**{** ALU\_Result**[**1**]** **&** **~**ALU\_Result**[**0**]}** **}** **&** **{** Read\_data**[**23**:** 0**]** **,** RF\_rdata2**[** 7**:** 0**]** **}** **)|**  **({** 32**{** ALU\_Result**[**1**]** **&** ALU\_Result**[**0**]}** **}** **&** Read\_data**[**31**:** 0**]** **)** **;**  **assign** lwr\_data **=** **({** 32**{~**ALU\_Result**[**1**]** **&** **~**ALU\_Result**[**0**]}** **}** **&** Read\_data**[**31**:** 0**]** **)|**  **({** 32**{~**ALU\_Result**[**1**]** **&** ALU\_Result**[**0**]}** **}** **&** **{** RF\_rdata2**[**31**:**24**]** **,** Read\_data**[**31**:** 8**]** **})|**  **({** 32**{** ALU\_Result**[**1**]** **&** **~**ALU\_Result**[**0**]}** **}** **&** **{** RF\_rdata2**[**31**:**16**]** **,** Read\_data**[**31**:**16**]** **})|**  **({** 32**{** ALU\_Result**[**1**]** **&** ALU\_Result**[**0**]}** **}** **&** **{** RF\_rdata2**[**31**:** 8**]** **,** Read\_data**[**31**:**24**]** **});**  **assign** load\_data **=** **(** **{**32**{**opcode**[**2**:**0**]==**3'b000**}}** **&** lb\_data **)** **|**  **(** **{**32**{**opcode**[**2**:**0**]==**3'b001**}}** **&** lh\_data **)** **|**  **(** **{**32**{**opcode**[**2**:**0**]==**3'b011**}}** **&** lw\_data **)** **|**  **(** **{**32**{**opcode**[**2**:**0**]==**3'b100**}}** **&** lbu\_data **)** **|**  **(** **{**32**{**opcode**[**2**:**0**]==**3'b101**}}** **&** lhu\_data **)** **|**  **(** **{**32**{**opcode**[**2**:**0**]==**3'b010**}}** **&** lwl\_data **)** **|**  **(** **{**32**{**opcode**[**2**:**0**]==**3'b110**}}** **&** lwr\_data **)** **;**    //store data rt->mem  //strb is signal showing which bytewrite is valid, code by truthtable //byte means vaddr/ALU\_result[2:0]  // vaddr swl\_strb swl\_shifter swr\_Strb swr\_shifter  // 00/0 0001 11000/24 1111 00000/0  // 01/1 0011 10000/16 1110 01000/8  // 10/2 0111 01000/8 1100 10000/16  // 11/3 1111 00000/0 1000 11000/24    **assign** sb\_strb **=** **{** ALU\_Result**[**1**]&** ALU\_Result**[**0**]** **,** ALU\_Result**[**1**]&** **~**ALU\_Result**[**0**]** **,** **~**ALU\_Result**[**1**]&** ALU\_Result**[**0**]** **,** **~**ALU\_Result**[**1**]&** **~**ALU\_Result**[**0**]};**  **assign** sh\_strb **=** **{** **{**2**{**ALU\_Result**[**1**]}}** **,** **{**2**{~**ALU\_Result**[**1**]}}** **}** **;**  **assign** sw\_strb **=** 4'b1111 **;**  **assign** swl\_strb **=** **{** **&**ALU\_Result**[**1**:**0**]** **,** ALU\_Result**[**1**]** **,** **|**ALU\_Result**[**1**:**0**]** **,** 1'b1**};**  **assign** swr\_strb **=** **{** 1'b1**,** **~&**ALU\_Result**[**1**:**0**]** **,** **~**ALU\_Result**[**1**],** **~|**ALU\_Result**[**1**:**0**]}** **;**  **assign** Write\_strb **=** **(** **{**4**{**opcode**[**2**:**0**]==**3'b000**}}** **&** sb\_strb **)** **|**  **(** **{**4**{**opcode**[**2**:**0**]==**3'b001**}}** **&** sh\_strb **)** **|**  **(** **{**4**{**opcode**[**2**:**0**]==**3'b011**}}** **&** sw\_strb **)** **|**  **(** **{**4**{**opcode**[**2**:**0**]==**3'b010**}}** **&** swl\_strb **)** **|**  **(** **{**4**{**opcode**[**2**:**0**]==**3'b110**}}** **&** swr\_strb **)** **;**    **assign** sb\_data **=** **{** 4**{**RF\_rdata2**[**7**:**0**]}** **}** **;**  **assign** sh\_data **=** **{** 2**{**RF\_rdata2**[**15**:**0**]}}** **;**  **assign** sw\_data **=** RF\_rdata2 **;**  **assign** swl\_shifter **=** **{~**ALU\_Result**[**1**:**0**]** **,** 3'b0**};**  **assign** swr\_shifter **=** **{** ALU\_Result**[**1**:**0**]** **,** 3'b0**};**  // Reuse shifter module to save two shifter  // swl\_data = RF\_rdata2 >> swl\_shifter ;  // swr\_data = RF\_rdata2 << swr\_shifter ;  **assign** Write\_data **=** **(** **{**32**{**opcode**[**2**:**0**]==**3'b000**}}** **&** sb\_data **)** **|**  **(** **{**32**{**opcode**[**2**:**0**]==**3'b001**}}** **&** sh\_data **)** **|**  **(** **{**32**{**opcode**[**2**:**0**]==**3'b011**}}** **&** sw\_data **)** **|**  **(** **{**32**{**opcode**[**1**:**0**]==**2'b10**}}** **&** Shifter\_Result **)** **;** |

Load和Store两类指令均包含不同指令，需对于不同指令确定数据。对load类指令，ALU\_Result的后两位确定了新数据覆盖原数据的位置；对store类指令，ALU\_Result的后两位确定了掩码，用以向内存传递新数据需要覆盖的位置。同时对于swl和swr指令，需要对RF\_rdata2进行移位操作，可复用Shifter以节约资源，且将需移动的位数用二进制表示可以逐位用ALU\_Result表示，减少资源消耗。Load和Store最后根据opcode对相应指令得到数据进行选择即可。

1. 多周期simple\_cpu

逻辑结构图：只含有PC+br\_extension一个加法器和ALU，只含有Shifter一个移位器。相比单周期处理器减少了PC+4加法器，在不同阶段间增加了寄存器。



省略变量端口声明和与单周期重复部分，只说明新增及修改代码，大体可分为状态机部分，寄存器部分，ALU部分和Shifter部分。

1. 状态机部分

|  |
| --- |
| //define the state of machine by one-hot  **localparam** RST **=**9'b000000001**,**  IF **=**9'b000000010**,**  ID **=**9'b000000100**,**  EX **=**9'b000001000**,**  MEM **=**9'b000010000**,**  WB **=**9'b000100000**;**  //State Machine  //Part I:  **always** **@** **(posedge** clk**)** **begin**  **if(**rst**)** **begin**  current\_state **<=** IF**;**  **end**  **else** **begin**  current\_state **<=** next\_state**;**  **end**  **end**  //Part II: Combinatorial logic  **always** **@** **(\*)**  **begin**  **case(**current\_state**)**  IF**:** **begin**  next\_state **=** ID **;**  **end**  //NOP: Instr = 32'b0  ID**:** **begin**  **if(~|**Instruction\_Reg**)**  next\_state **=** IF **;**  **else**  next\_state **=** EX **;**  **end**  //EX->IF:  //REGIMM 2: opcode[5:0]=000001  //I\_branch 4: opcode[5:2]=0001  //J 1: opcode[5:0]=000010  //EX->WB  //R\_Type 18: opcode[5:0]=6'b0 include JALR,JR //JR: rs->GPR[0]  //I\_calc 7: opcode[5:3]=001 include LUI  //JAL 1: opcode[5:0]=000011  //EX->MEM  //load and store 12: opcode[5]=1  EX**:** **begin**  **if((**opcode**[**5**:**0**]==**6'b000001**)** **|** **(**opcode**[**5**:**2**]==**4'b0001**)** **|** **(**opcode**[**5**:**0**]==**6'b000010**)** **)**  next\_state **=** IF**;**  **else** **if((~|**opcode**)|** **(**opcode**[**5**:**3**]==**3'b001**)** **|** **(**opcode**[**5**:**0**]==**6'b000011**))**  next\_state **=** WB**;**  **else** **if(**opcode**[**5**]==**1'b1**)**  next\_state **=** MEM**;**  **end**  //MEM->IF: store: opcode[5:3]=101  //MEM->WB: load : opcode[5:3]=100  MEM**:** **begin**  **if(**opcode**[**5**:**3**]==**3'b101**)**  next\_state **=** IF**;**  **else** **if(**opcode**[**5**:**3**]==**3'b100**)**  next\_state **=** WB**;**  **end**  WB**:** **begin**  next\_state **=** IF**;**  **end**  **default:** **begin**  next\_state **=** IF**;**  **end**  **endcase**  **End** |

状态机的第一部分使用时序逻辑进行现态的更新，第二部分使用组合逻辑根据现态和指令确定次态情况。转移条件如代码注释所示。特别的，ID阶段需要考虑空指令的跳转。

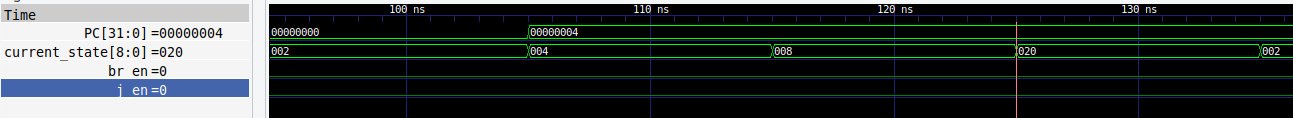
1. 寄存器部分

说明：ALU和Shift输入输出相关寄存器见其对应部分

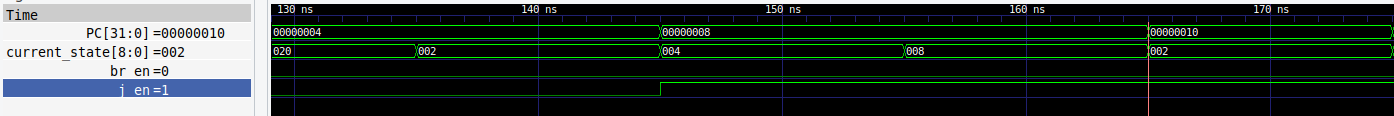
|  |
| --- |
| //Part III: deal with output  //one always module to deal one reg  **always** **@** **(posedge** clk**)** **begin**  **if(**current\_state**[**1**])** //IF  Instruction\_Reg **<=** Instruction **;**  **end**    **always** **@** **(posedge** clk**)** **begin**  **if(**current\_state**[**4**])** //MEM  Read\_data\_Reg **<=**Read\_data **;**  **end**    **always** **@** **(posedge** clk**)** **begin**  **if(**current\_state**[**1**])** //IF  PC\_Reg **<=** PC**;**  **end**    **always** **@** **(posedge** clk**)** **begin**  **if** **(**rst**)** **begin**  PC **<=** 32'b0**;**  **end**  **else** **if(**current\_state**[**1**]** **&** **~**rst**)** **begin**  PC **<=** ALU\_Result**;**  //all OP refresh in ID, judge by IF  **end**  **else** **if(**current\_state**[**3**]** **&** **~**rst **&(|**Instruction\_Reg**))begin**  PC **<=** br\_en **?** br\_tar **:** j\_en **?** j\_tar **:** PC **;**  //j or br OP refresh in next IF, judge by EX  //note that the default result is PC  **end**  **end** |

由于PC、Instruction和Read\_data均只维持一拍，多周期CPU新增了相应寄存器将这些值进行寄存。而多周期一条指令过程中PC将更新两次，第一次在IF阶段对所有指令PC均更新加4，第二次在EX阶段根据ALU运算结果和指令类别确定是否条件跳转或无条件跳转，否则维持PC不变。

相应波形说明：



当不产生跳转时，PC只在IF阶段(002)后更新一次，PC+4;



当产生跳转时，PC在IF阶段(002)后更新一次，PC+4;在EX阶段(008)后再更新一次，跳转到目标值。

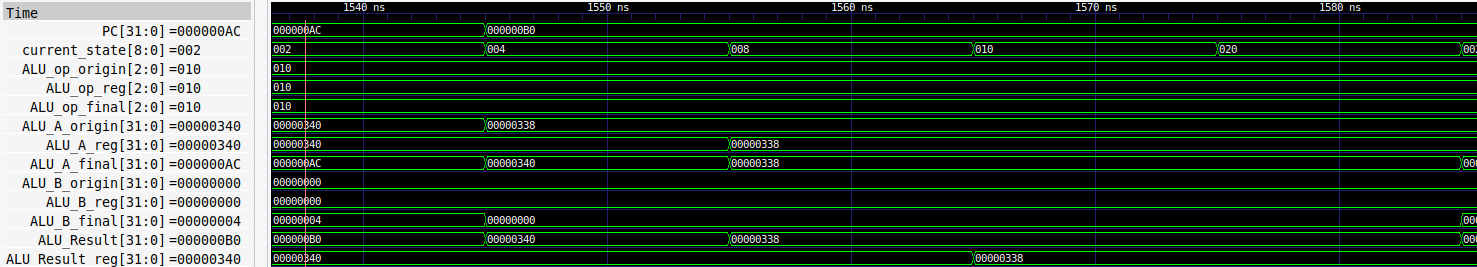
1. ALU部分

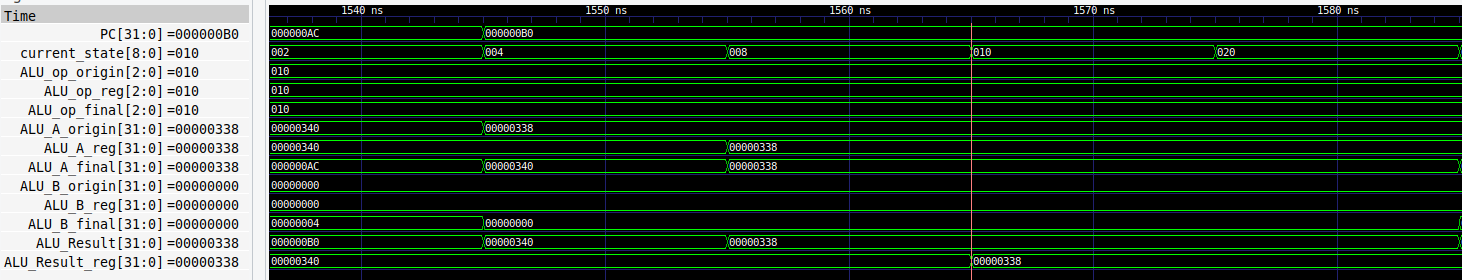
|  |
| --- |
| //signals connected to ALU  //IF stage: All instruction: PC+4  //Other stage: To save adder, reuse ALU when PC+8 (swl swr)  //All type below can differ from others  //Operation related to ALU list as : Type (op\_A op\_B) num : feature  //R calc (rs rt) 8 : opcode[5:0]=000000 & func[5]=1  //I calc (rs imm) 6 : opcode[5:3]=001 & (~&opcode[2:0]) diff from lui  //REGIMM (rs 0) 2 : opcode[5:0]=000001 BGEZ BLTZ GPR[rs]?0 using SLT(sign)  //Branch (rs rt) 2 : opcode[5:1]=00010  //(rs 0) 2 : opcode[5:1]=00011 using SUB and test ZERO or signflag  //load and store (rs imm) 12 : opcode[5]=1 using ADD  //base(rs)+offset(imm)  //JAL (PC\_reg 8) 1: opcode[5:0]=000011  //JALR (PC\_reg 8) 1: opcode[5:0]=000000 & func[5:1]=001001  //Operation not related to ALU list:  //shifter 6 : opcode[5:0]=000000 & func[5:3]=000  //jump 2 : J opcode[5:0]=000010  //JR opcode[5:0]=000000 & func[5:0]=001000  //lui 1 : opcode[5:0]=001111  //move (rt 0) 2 : opcode[5:0]=000000 & func[5:1]=00101  //move rs to rd,  //we can use ALU(ADD/SUB) or no, and we should care the condition about Reg\_wen  **assign** ALU\_op\_origin **=** **(~|**opcode**[**5**:**0**])** **&** func**[**5**]==**1'b1 **?** **(**func**[**3**:**2**]==**2'b00 **?** **{**func**[**1**],**2'b10**}:**  func**[**3**:**2**]==**2'b01 **?** **{**func**[**1**],**1'b0**,**func**[**0**]}** **:**  func**[**3**:**2**]==**2'b10 **?** **{~**func**[**0**],**2'b11**}** **:**  0 **)** **:**  opcode**[**5**:**3**]==**3'b001 **&** **(~&**opcode**[**2**:**0**]) ?** **(**opcode**[**2**:**1**]==**2'b00 **?** **{**opcode**[**1**],**2'b10**}** **:**  opcode**[**2**]==**1'b1 **?** **{**opcode**[**1**],**1'b0**,**opcode**[**0**]}** **:**  opcode**[**2**:**1**]==**2'b01 **?** **{~**opcode**[**0**],**2'b11**}** **:**  0 **)** **:**  opcode**[**5**:**0**]==**6'b000001 **?** 3'b111 **:**  opcode**[**5**:**2**]==**4'b0001 **?** 3'b110 **:**  opcode**[**5**]** **==**1'b1 **?** 3'b010 **:**  opcode**[**5**:**0**]==**6'b000011 **|** **((~|**opcode**[**5**:**0**])** **&** func**[**5**:**0**]==** 6'b001001**)** **?** 3'b010 **:**  0**;** //default    **always** **@(posedge** clk**)** **begin**  **if(**rst**)**  ALU\_op\_reg **<=**0**;**  **else** **if(**current\_state**[**2**])** //ID  ALU\_op\_reg **<=** ALU\_op\_origin**;**  **else**  ALU\_op\_reg **<=** ALU\_op\_reg**;**  **end**  **assign** ALU\_op\_final **=** current\_state**[**1**]** **?** 3'b010 **:** ALU\_op\_reg**;**    **assign** ALU\_A\_origin **=** opcode**[**5**:**0**]==**6'b000011 **|** **((~|**opcode**[**5**:**0**])** **&** func**[**5**:**0**]==**6'b001001**)** **?** PC\_Reg **:**  RF\_rdata1**;**  **always** **@(posedge** clk**)** **begin**  **if(**rst**)**  ALU\_A\_reg **<=** 0**;**  **else** **if(**current\_state**[**2**])**//ID  ALU\_A\_reg **<=** ALU\_A\_origin**;**  **else**  ALU\_A\_reg **<=** ALU\_A\_reg**;**  **end**  **assign** ALU\_A\_final **=** current\_state**[**1**]** **?** PC **:** ALU\_A\_reg**;**    //ALU\_B: 4 / 8/ rt / 0 / sign\_extend(imm) /zero\_extend(imm)  //imm\_extension: I\_calc 6 + load and store 12  //zero\_extension : ANDI ORI XORI opcode[2]=1  //sign\_extension : ADDIU SLTI SLTIU opcode[2]=0 load and store: opcode[5]==1  **assign** ALU\_B\_origin **=** opcode**[**5**:**3**]==**3'b001 **&** **(~&**opcode**[**2**:**0**])** **?** **(** opcode**[**2**]** **?** zero\_extension **:** sign\_extension **)** **:**  opcode**[**5**]** **?** sign\_extension **:**  opcode**[**5**:**0**]==**6'b000001 **?** 32'b0 **:**  opcode**[**5**:**0**]==**6'b000011 **|** **((~|**opcode**[**5**:**0**])** **&** func**[**5**:**0**]==** 6'b001001**)** **?** 32'd8 **:**  RF\_rdata2**;**  **always** **@(posedge** clk**)** **begin**  **if(**rst**)**  ALU\_B\_reg **<=** 0**;**  **else** **if(**current\_state**[**2**])**  ALU\_B\_reg **<=** ALU\_B\_origin**;**  **else**  ALU\_B\_reg **<=** ALU\_B\_reg**;**  **end**  **assign** ALU\_B\_final **=** current\_state**[**1**]** **?** 32'd4 **:** ALU\_B\_reg**;**  **always** **@(posedge** clk**)** **begin**  **if(**rst**)**  ALU\_Result\_reg **<=** 0**;**  **else** **if(**current\_state**[**3**])**//EX  ALU\_Result\_reg **<=** ALU\_Result**;**  **else**  ALU\_Result\_reg **<=** ALU\_Result\_reg**;**  **end**  alu alu\_inst**(**  **.**A**(**ALU\_A\_final**),**  **.**B**(**ALU\_B\_final**),**  **.**ALUop**(**ALU\_op\_final**),**  **.**Overflow**(**ALU\_Overflow**),**  **.**CarryOut**(**ALU\_CarryOut**),**  **.**Zero**(**ALU\_Zero**),**  **.**Result**(**ALU\_Result**)**  **);** |

对于多周期下的ALU，单周期相应的操作只在EX阶段进行，其余阶段闲置，可在这些阶段被调用以减少加法器数量、节约电路资源。

对于单周期相应操作，设置ALU\_X\_origin信号，在EX的前一阶段ID将其寄存，并将结果在EX阶段进行寄存。同时考虑到ALU可在IF阶段复用以得到PC+4，因此设计数据旁路，通过选择数据旁路值和寄存器得到最终的ALU操作数和控制信号。寄存器和数据旁路的设计可以使得ALU兼具时效性、稳定性和复用程度高的特点。

相应波形说明：





该条指令在IF阶段(002)使用了数据旁路，复用ALU获得了PC+4的值，在ID阶段(004)得到了ALU操作数，在EX阶段(008)操作数寄存值更新得到运算结果，并在后一阶段得到结果的寄存值。

1. Shifter部分

|  |
| --- |
| //signals connected to Shifter  //Operation related to Shifter list: 6 + 2(swl swr)  //B width is 5  //R shift: opcode[5:0]=000000 & funct[5:3]=000  //xxx (rt sa) 3: fun[2]=0  //xxxv (rt rs) 3: fun[2]=1  //Swl/swr: opcode[5:0]=101x10  //swl/swr (rt swl\_shifter)  **assign** Shifter\_op **=** **(~|**opcode**[**5**:**0**])&(~|**func**[**5**:**3**])** **?** func**[**1**:**0**]** **:**  **{**opcode**[**5**:**3**],**opcode**[**1**:**0**]}** **==** 5'b10110 **?** **{~**opcode**[**2**],**1'b0**}** **:**  0**;**  **always** **@(posedge** clk**)** **begin**  **if(**current\_state**[**2**])**  Shifter\_op\_reg **<=** Shifter\_op**;**  **end**  **assign** Shifter\_A **=** RF\_rdata2**;**  **always** **@(posedge** clk**)** **begin**  **if(**current\_state**[**2**])**  Shifter\_A\_reg **<=** Shifter\_A**;**  **end**  **assign** Shifter\_B **=** **(~|**opcode**[**5**:**0**])&(~|**func**[**5**:**3**])** **?** **(**func**[**2**]** **?** RF\_rdata1**[**4**:**0**]** **:** sa**)** **:** 0 **;**  **always** **@(posedge** clk**)** **begin**  **if(**current\_state**[**2**])**  Shifter\_B\_reg **<=** Shifter\_B**;**  **end**  //reuse shifter by swx in MEM  **assign** Shifter\_B\_final **=**  current\_state**[**4**]** **&** **{**opcode**[**5**:**3**],**opcode**[**1**:**0**]}** **==** 5'b10110 **?**  **(**opcode**[**2**]** **?** swr\_shifter **:** swl\_shifter**)** **:**  Shifter\_B\_reg**;**    **always** **@(posedge** clk**)** **begin**  **if(**rst**)**  Shifter\_Result\_reg **<=** 0**;**  **else** **if(**current\_state**[**3**])**  Shifter\_Result\_reg **<=** Shifter\_Result**;**  **end**  shifter shifter\_inst**(**  **.**A**(**Shifter\_A\_reg**),**  **.**B**(**Shifter\_B\_final**),**  **.**Shiftop**(**Shifter\_op\_reg**),**  **.**Result**(**Shifter\_Result**)**  **);** |

与ALU相似，Shifter也可在闲置阶段被复用以节约资源。此处同样设置origin处理单周期中的运算指令，并设置数据旁路处理swl和swr指令所需的数据移位操作。

1. 实验过程中遇到的问题、对问题的思考过程及解决方法（比如RTL代码中出现的逻辑bug，逻辑仿真和FPGA调试过程中的难点等）

问题1：Verilog位运算语法问题

{}外加数字进行位拓展应当在外侧再加一对{}，查看云平台语法检查log发现问题更正。

问题2：电路资源精简

ALU和Shifter可以在闲置时被复用以节约资源，如PC+8的加法操作和swl、swr的移位操作都可以进行复用，同时在多周期CPU中，由于IF阶段ALU闲置，可以被复用以进行PC+4操作。

问题3：golden比对问题

在提前进行实验过程中发现golden状态更新逻辑与多周期状态机不符，进行反馈后更正。

问题4：多周期CPU的寄存器使用和协同问题

验收阶段和助教及老师的交流过程中，我理解了ALU、Shifter的输入输出寄存器可以将不同状态阶段进行分隔，从而减少最长组合逻辑链长度，允许处理器运行更高的频率。

由于PC+4对ALU的复用和swl、swr对Shifter的复用均只在当拍内，无法使用寄存器处理，因此可以设计数据旁路，并在输入ALU时选择数据旁路或寄存器，完成寄存器和数据旁路的协同，从而同时具备节约资源和降低延迟的优势。对于ALU和Shifter的输出，在本状态阶段内使用输出立即值，在不同状态阶段的使用寄存值，使得数据可以具有时效性和稳定性。

问题5：多周期CPU添加寄存器时错误

Debug过程：ALU\_Result错误 <- ALU\_A\_origin 错误 <- RF\_rdata1未定义 <- 对应addr的RF\_wdata未定义 <- 写寄存器不同情形 <- Shifter\_Result错误 <- Shifter\_B\_final default态错误。通过错误逻辑链向前追溯到错误发生处进行更正。

1. 对讲义中思考题（如有）的理解和回答

ALUop编码规律和优化

编码规律：

R-type由最低4位映射出ALUop，I-type由最低3位映射出ALUop，二者均可以通过两位func或opcode划分为加减运算、逻辑运算和比较运算三类。

R-type和I-type对应指令的最低三位一致（除lui外），因此在对应分类中可以使用相同的编码规则。

优化：

由先分类后编码的粗粒度编码转化为逐位分类编码的细粒度编码，减少了选择层数，代码如下所示：

|  |
| --- |
| R-type：  ALUop[2] = (~func[3]&func[1])|(func[3]&~func[0])  ALUop[1] = ~func[2]  ALUop[0] = (func[2]&func[0])|(~func[2]&func[3])  I-type:  ALUop[2] = (~opcode[2]&opcode[1]&~opcode[0])|  ((opcode[2]|~opcode[1])&opcode[1])  ALUop[1] = ~opcode[2]  ALUop[0] = (~opcode[2]&opcode[1])|(opcode[2]&opcode[0]) |

1. 在课后，你花费了大约\_\_\_20\_\_\_小时完成此次实验。
2. 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

心得感受：

本次实验单周期处理器部分考察的主要仍是对指令集的理解，对Verilog代码并未具有太高要求，代码编写过程中主要考察的是对指令的理解归纳以及将逻辑相关代码分模块编写的能力。

多周期处理器部分对时序逻辑的理解和CPU处理指令的流程提出了更高要求。一方面需要对只能维持一拍的数据进行寄存，且需关注数据在哪个状态阶段应有效，另一方面需要根据指令确定状态转移规则。同时在验收过程中通过和助教、老师的交流我理解了寄存器在不同阶段中起到的作用（详见第二部分），以及寄存器和数据旁路的协同（如PC+4临时借用ALU），这对后续prj3和pipeline数据传递的处理具有很大帮助。

建议：

在提前进行实验的过程中我发现多周期golden文件的PC更新逻辑问题，后续老师进行了修正更新。但由于PC更新逻辑可以具有多种写法，而仿真阶段的golden写法是确定的，这就可能产生上板可以正常运行但是仿真出错的现象。为支持多种写法，可通过编写多个不同更新逻辑的golden文件，在bhv\_sim阶段使用多个同名任务，或发起多个pipeline解决，有一个通过即为通过。

致谢：

感谢陈欲晓助教和张老师在验收过程中对寄存器作用的深入讲解，感谢芦溶民助教和常老师在golden更新和云平台相关问题的解答和帮助。